Dimensional Analysis and Inference for gPROMS

B. Daniel Persson
Master of Science thesis
Department of Computer Science and Engineering
Mälardalen University
dpn99004@student.mdh.se

Supervisor: Mikael Sandberg
Supervisor, ABB: Tomas Lindberg

Examiner: Prof. Björn Lisper

October 8, 2003

Abstract

In this thesis we describe a dimensional analysis and inference system for a strongly typed language for simulation of physical systems, gPROMS. We show how gPROMS can benefit from such a system and we believe that it will increase the physical correctness of the simulation models.

Our proposed dimensional inference system is based on generating dimensional constraint equations from the model equations using formal inference rules. The resulting constraints constitute a linear system of equations which is solved by means of linear algebra. If the system of equations is solvable the simulation model is proven to be dimensionally consistent.

We have implemented our dimensional inference algorithms in Java in the form of a stand-alone tool. The tool will be used by the industry to quickly locate physical errors in simulation models resulting from dimensionally inconsistent equations.

Contents

List of Figures

List of Algorithms

Acknowledgments

This thesis is the result of a joint venture between the Department of Computer Science and Engineering at Mälardalen University and the Department of Industrial IT at ABB Corporate Research in Västerås.

I would like to thank ABB Corporate Research for their financial support and my supervisors Mikael Sandberg and Tomas Lindberg for many rewarding discussions. Also, many thanks to the members of the programming language group in the Computer Science Lab at Mälardalen University, especially Björn Lisper, Jan Carlson and Marcus Bohlin.

Chapter 1

Background

Usually when developing simulation models, there is no way of automatically telling whether the models are correct or not. But, since the models are specified in a formal computer readable syntax, automatic analysis can be performed statically to verify certain properties.

One interesting property when simulating physical systems is the *dimensional consistency* of the equations that make up the system. For instance, one can never add or subtract values of different dimensions, and the multiplication or division of two values yields results whose dimensions are also multiplied or divided. Dimensional consistency can be verified using a method known as *dimensional analysis* (DA). In using DA, it will be possible to detect physical inconsistencies in the simulation model at an early stage of development, reducing the development time.

In order for a system to be as versatile as possible, it is of great importance to allow for polymorphic models. This can be achieved through *type inference* for dimension types, also known as *dimensional inference* (DI). DI can be used to automatically infer the missing dimensions in the model specifications. The missing dimensions can either be intentional, in case of polymorphic models, or unintentional, in case of legacy models or models not developed in-house. In either case, the system must derive the missing dimensions from the information that is given, which may result in dimensional inconsistencies. The advantage with a DI system is that more models will be possible to process without the need for extensive annotation of dimensions.

1.1 Purpose

The purpose of this thesis is to develop a platform for evaluation of dimensional analysis and dimensional inference for simulation languages. The platform will be in the form of a stand-alone tool for dimensional analysis and dimensional inference for gPROMS, a general PROcess Modeling System. The tool will provide a quantitative measurement on the dimensional correctness of the models, and it will also aid in developing large scale simulation models by reducing the amount of time normally spent on verification and testing.

1.2 Outline

Chapter 2 gives an introduction to the gPROMS language and different areas regarding type systems and dimensional analysis. Moving on to Chapter 3 we describe our proposed dimensional inference system and discuss some of the issues that have

been encountered. Chapter 4 gives a detailed explanation of our dimensional inference system for gPROMS. Chapter 5 shows some results from tests on fictitious models. In Chapter 6 we will discuss some of the limitations of our system and how it could be improved in the future. Finally, chapter 7 concludes the thesis with a summary.

Chapter 2

Introduction

In this chapter we will introduce the gPROMS language and its applicability in the industry. We will also describe the background of dimensional analysis and how it is used to derive dimensional consistency. The chapter ends with a related work section where we review previous work in dimensional analysis and inference for programming languages.

2.1 Introduction to gPROMS

gPROMS is a programming language for simulation, optimization and parameter estimation of highly complex processes developed at the Center for Process Systems Engineering at Imperial College. The user specifies the equations that make up the simulation model using a simple syntax with no concern for the complexity of the solution techniques. The equations are transformed into a set of Partial Differential Algebraic Equations (PDAE's), which are solved using an advanced numerical solver. The solvers are designed specifically for large-scale systems and there are no limits regarding the size of the simulation other than those imposed by the machine running the system.

gPROMS models complex processes with operating procedures using MODELs and TASKs. MODELs are used to describe the physical, chemical and biological behavior of the process. The TASKs describe the actual operating procedure that is used to run a process, and operates on MODELs. The gPROMS TASK syntax is very general and allows description of complex operating procedures, each comprising a number of steps to be executed in sequence, parallel, conditionally or iteratively.

gPROMS has the ability to model discontinuous processes, where changes take place abruptly and frequently due to phase transitions, flow transitions, geometrical limitations and so forth. The system allows the direct mathematical description of distributed unit operations where properties vary in one or more spatial dimensions. The generality of gPROMS means that it can be used for a wide variety of applications in petrochemicals, food, pharmaceutical, specialty chemicals and automation.

2.2 Type Systems and Type Inference

Type systems are used in programming languages to prevent the occurrences of certain execution errors, referred to as *type errors*. A language is said to be *type sound* if for all possible programs that can be expressed within the language there will be no type errors.

A formal type system is based on a collection of *type rules*. The type rules determine how types can be derived in the system. If there is no possible type derivation for a term, we have a type error. In contrast, if a derivation exists the term is given the type of the derivation, which is not necessarily unique.

Type inference enables programs that lacks type information to be safely typed, by automatically infer the most general types. Many of the functional languages, such as ML [MTH89] and Haskell [Jon99, JH99] use type inference. To infer a missing type, a derivation must be discovered using the type rules of the type system. The algorithms involved in this process range from relatively simple to quite complicated, depending on the underlying type system. It might even be impossible to find an inference algorithm [Car96].

2.3 Dimensional Analysis

Dimensional analysis is a technique that has been used by physicists and engineers for many years to obtain preliminary solutions to physical problems. It is very useful for derivation of an analytical solution when the variables that take part in a physical phenomenon are known, but the way they relate to each other is unknown. DA originates from the work of Newton [New87] and Fourier [Fou22] and has since been refined by several scientists, in particular Rayleigh [Ray15, Ray78] and Buckingham [Buc14] among others.

Fourier introduced the notion of a dimensional formula and showed that equations should have *dimensional homogeneity*. A more formal description of dimensional homogeneity was formed by Buckingham and the Π-theorem [Buc14], which is the main theorem of DA. It states that a physical law relating a set of variables can be expressed as a function of a lesser number of dimensionless arguments, called Π-groups. Each Π-group is a dimensionless product of a combination of integer powers of the original variables. The general form of the solution is determined by finding the function relating the Π-groups, which is usually derived experimentally.

Following the Π-theorem, any formula must be dimensionally homogeneous in order to have any physical meaning. If both sides of an equation are not of the same dimension, we have a dimensional inconsistency, which leads to the use of DA as a dimensional consistency check.

2.3.1 Dimensions and Units

Dimensions describe the properties of physical quantities, be it length, mass, force etc. The different physical quantities are measured in reference to a *system of units*. Two quantities with different units but the same dimension differ only by a scaling factor. If the scaling factor is a constant we say that the two quantities are *commensurate* [KL78]. There are more complicated units that do not have simple conversions such as temperature measured in Celsius and Fahrenheit or worse, amplitude level in decibels.

We have *base dimensions* and *derived dimensions*. Base dimensions cannot be defined in terms of other dimensions. The International System of Units (SI) defines seven base dimensions — length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity. Derived dimensions are defined in terms of base dimensions. For instance, velocity is distance divided by time. The seven base dimensions defined in the SI system are represented by the units — meters, kilograms, seconds, Amperes, Kelvin, moles and Candela. SI also defines 22 derived dimensions that are given special names, for example Newtons (N) $[kg \cdot m/s^2]$ and Hertz (Hz) $[s^{-1}]$.

2.3.2 Dimensional Algebra

Dimensions satisfies the algebraic properties of an abelian group whose operation is dimension product [Ken96]. For instance, the dimension $[M \cdot T^{-1}]$ is equivalent to $[T^{-1} \cdot M]$, and $[M \cdot M^{-1}]$ is equivalent to the unit dimension $\mathbf{1}$. Figure 2.1 shows the properties of abelian groups.

$$\begin{array}{rcll} d_1 \cdot d_2 & = & d_2 \cdot d_1 & \textit{commutativity} \\ (d_1 \cdot d_2) \cdot d_3 & = & d_1 \cdot (d_2 \cdot d_3) & \textit{associativity} \\ \mathbf{1} \cdot d & = & d & \textit{identity} \\ d \cdot d^{-1} & = & \mathbf{1} & \textit{inverses} \end{array}$$

Figure 2.1: Abelian group properties

Much like the addition, subtraction and comparison of values of different types is a type error, the same is true for dimension types. For example, you cannot add meters to seconds and regard the result as physically meaningful. On the other hand, when multiplying or dividing quantities of any dimensions the result receives a dimension which is the product or quotient of the dimensions of the two quantities. Following this argument, the sum of two values with dimensions velocity $[L \cdot T^{-1}]$ and time $[T]$ is a dimension error, whereas their product has dimension length $[L]$.

Dimensions are often represented as vectors of the base dimensions exponents. For example, if we have three base dimensions, MLT ($Mass$, $Length$ and $Time$), the dimension $Force$ $[M \cdot L \cdot T^{-2}]$ can be represented as $\langle 1, 1, -2 \rangle$. In this vector form, the multiplication of two dimensions is equivalent to vector addition, and division is equivalent to vector subtraction. The unit dimension $\mathbf{1}$ is represented as the zero vector $\vec{0}$. The dimension of a variable is represented with δ_{name}. For example, consider a physical equation calculating velocity as length divided by time:

$$v = \frac{l}{t}$$

The dimensions of the variables v, l, t are represented by the symbols δ_v, δ_l and δ_t. δ_v represents $Velocity$ $\langle 0, 1, -1 \rangle$, δ_l represents $Length$ $\langle 0, 1, 0 \rangle$ and δ_t represents $Time$ $\langle 0, 0, 1 \rangle$. Given this information we can check the consistency of the equation by examining both sides of the equation, using the algebraic rules for dimensions. The dimensional consistency of the equation above is derived as follows:

$$\begin{array}{rcl} \delta_l & = & \langle 0, 1, 0 \rangle \\ \delta_t & = & \langle 0, 0, 1 \rangle \\ \delta_v & = & \delta_l / \delta_t = \\ & & \langle 0, 1, 0 \rangle - \langle 0, 0, 1 \rangle = \\ & & \langle 0, 1, -1 \rangle = \delta_v \end{array}$$

2.4 Related Work

Extensive effort has been put into researching the area of dimensional analysis for programming languages. A lot of articles relating to the subject have been studied in order to grasp certain techniques used in both dimensional analysis and dimensional inference. This includes dimensional analysis, dimensional inference, unit checking and unit conversion.

2.4.1 Dimensional Analysis

Dimensional analysis has been implemented for numerous different general programming languages — Ada [Hil88], ML [Ken94, WO91], Pascal [Hou83], C++ [CG88, BN95], LISP [Nov95] among others. Also, Khanin has implemented a dimensional analysis package for Mathematica [Kha01]. However, only little has been done regarding dimensional analysis for physical simulation languages, such as Modelica and gPROMS.

House [Hou83] was one of the first to implement a polymorphic dimensional analysis system for a monomorphic language, Pascal. He used a construction, *newdim*, which allowed the definition of dimensionally polymorphic arguments for procedures.

Cmelik and Gehani [CG88] uses the class abstraction facilities of C++ to automatically perform dimensional analysis at runtime. Barton and Nackman [BN95] on the other hand uses templates along with classes to define a system that checks the program statically.

Kennedy [Ken94] proposed a dimensional inference system based on ML–style type inference and equational unification. His system uses integer exponents to represent dimensions, which restricts the expressiveness. It is not possible, for example, to take the square root of Length. Kennedy argues that fractional exponents are rarely seen in science and if such a thing arose it would suggest a revision of the set of base dimensions [Ken94, Ken97].

A similar system to that of Kennedy was proposed by Wand and O'Keefe [WO91]. The difference is that they represent dimensions using rational exponents to allow for more general dimensional expressions. Their system generates equations over dimensions, which are solved using gaussian elimination.

2.4.2 Unit Checking and Unit Conversion

Some work has also been done in the field of unit checking and automatic unit conversion, which is very similar to dimensional analysis. It seems like most implementations can only handle conversions between commensurate units. This is not sufficient for a physical simulation system, since temperature is an important factor.

Karr and Loveman [KL78] describes a method for finding conversion factors between commensurate units, which involves logarithms, matrix manipulations and linear algebra. They also present some details regarding an actual implementation of a unit checking system.

Gordon Novak [Nov95] implemented a unit checking and unit conversion system for LISP. His system checks that the expressions are dimensionally consistent and automatically converts commensurable units when necessary.

Hilfinger's dimensional analysis package for Ada can handle conversions between commensurate units [Hil88]. It uses Ada's strong typing together with its abstraction facilities and operator overloading to statically perform the dimensional consistency check. Also, he proposes some optimizations to the compiler in order to make the algorithm more effective.

Cmelik and Gehani's implementation for C++ can convert commensurate units at runtime [CG88].

Chapter 3

gPROMS

gPROMS is a strongly typed modeling language for simulation and optimization of physical systems. These systems are often quite large and complex and therefore difficult to model. gPROMS has an object based approach to modeling that enables hierarchical sub-model decomposition which will reduce the overall complexity and increase model reuse. This is not to say that gPROMS is an object oriented language, but rather incorporates some object oriented features.

Simulation models in gPROMS are defined using four different entities; `DECLARE`, `MODEL`, `TASK` and `PROCESS`. The `DECLARE` entity is used to declare variable types and stream types, which will be used as templates for variables and streams. The `MODEL` entity is used to describe the physical behavior of primitive elements of the system being modeled. A system usually consists of several different `MODELs` that are interconnected to form a highly complex simulation model. A `TASK` describes a certain operating procedure such as opening a valve or switching on a pump. The task concept is very similar to a procedure in a normal programming language. The `PROCESS` entity is used to define the behavior of the dynamic simulation, such as setting numeric solution parameters and initiating model parameters. A complete EBNF syntax of the gPROMS language is shown in appendix A.

3.1 Declare

The `DECLARE` entity is used to declare variable types and stream types. These types are later used to declare variables and streams in `MODELs`. There are two different sections within a `DECLARE` entity; `TYPE` and `STREAM`. Figure 3.1 shows an example of a `DECLARE` entity.

```
DECLARE
  TYPE
    Temperature =   0.0 :   0.0 : 1E10  UNIT = "K"
    Energy      =   1.0 : -1E10 : 1E10  UNIT = "J"
    FlowRate    =   1.0 : -1E10 : 1E10

  STREAM
    InputStream  IS Temperature, Energy
    OutputStream IS Temperature, Energy
END
```

Figure 3.1: A simple `DECLARE` entity

Type

The TYPE section is used to define new variable types. Each variable type is associated with a default value along with a value range. It is also possible to specify a unit of measure to accompany the variable type. This unit of measure is however only used when viewing graphs over the specific variable, where it is displayed along with the quantity.

Stream

The STREAM section is used to declare stream types, which is an ordered list of variable types. A stream is used to interconnect instances of variables in different models.

3.2 Model

In gPROMS, primitive models are declared via the MODEL entity. A MODEL contains a mathematical description of the physical behavior of a given system. It comprises a number of sections, each containing a different type of information regarding the system being modeled. The different sections are PARAMETER, DISTRIBUTION_DOMAIN, UNIT, VARIABLE, STREAM, SELECTOR, SET, BOUNDARY and EQUATION. Figure 3.2 shows a primitive model of a water cooler.

```
MODEL Cooler
  PARAMETER
    Cp   AS REAL        # Cooling capacity
    Rho  AS REAL        # Coolant Density
    V    AS REAL        # Cooler Volume

  VARIABLE
    Tin  AS Temperature # Coolant inlet temperature
    T    AS Temperature # Coolant temperature
    Q    AS Energy      # Heat load
    F    AS FlowRate    # Coolant flowrate

  EQUATION
    $T*V*Rho*Cp = (Tin - T)*Cp*F + Q; # dT/dt V Rho Cp = (Tin - T) Cp F + Q
END
```

Figure 3.2: A simple MODEL entity

Parameter

The PARAMETER section is used to declare parameters of a MODEL. Parameters are time-invariant quantities that will not be the result of any calculation. They can be thought of as constants in a normal programming language. The parameters are declared to be of one of the basic types, *integer*, *real*, *logical* or distributions of these. There is also another parameter type, the *foreign object* (FO). A foreign object is an external package that describes some physical properties. The foreign objects contain methods that are invoked from within a MODEL or TASK.

Distribution_domain

The DISTRIBUTION_DOMAIN section is used to declare the distribution domains of a MODEL. A distribution domain is used to declare variables that vary with respect to one or more intervals.

Unit

The `UNIT` section is used to declare the units of a `MODEL`. A unit is a submodel contained within a `MODEL`. The units can also be declared as arrays of `MODELs` whose size is determined by a parameter. In this way systems with a number of components of the same type can easily be parameterized.

Variable

The `VARIABLE` section is used to declare the variables of a `MODEL`. These represent quantities that describe the time-dependent behavior of a system. Each variable is declared to be of a specific variable type.

Stream

The `STREAM` section is used to declare the streams of a `MODEL`. A stream is a subset of the variables in a `MODEL` and provide a convenient mechanism for describing complex connections between components in a physical system. The stream type determines which variable types that can be contained within a specific stream.

The streams are only syntactic sugar for equivalence relationships between variables in different components. If two streams of the same type are connected, the system will generate equivalence equations between the corresponding variables.

Selector

The `SELECTOR` section is used to declare the selectors of a `MODEL`. gPROMS models discontinuous processes using *State-Transition Networks* (STNs) where each state contains a set of equations describing the physical behavior in that particular state. The selectors define the points where state transitions should occur according to the discontinuous system being modeled. Examples of discontinuous processes are laminar and turbulent flow, reversal of direction of flow, equipment failure etc.

Set

The `SET` section is used to declare initial assignments for parameters which will have effect before the simulation starts. The initiation is done via an `assign-stmnt`.

Boundary

The `BOUNDARY` section is used to declare boundary equations of a `MODEL`. For example, the boundary equations are used to describe boundary conditions for distributions.

Equation

The `EQUATION` section is used to declare the equations of a `MODEL`. The equations are mathematical descriptions of the physical behavior of the system being modeled. gPROMS extends the idea of an equation to include iterative and conditional equations, similar to elements of a normal imperative programming languages. We will refer to these equations as statements. The most trivial statement is the `equation-stmnt`. This statement describes the mathematical equations, entered as an equivalence relationship between general expressions. These expressions are formed using primitive arithmetic operations along with some language specific operations, such as `partial` and `integral`. Above the `equation-stmnt` are the following statements: `if-stmnt`, `for-stmnt` and `case-stmnt`, where `case-stmnt` is similar to the `switch-case` construction in C.

3.3 Task

TASKs are used to define operating procedures and they share some similarities with functions in normal programming languages. A TASK applies equations defined in an operating schedule on its arguments. The arguments can be general expressions or MODEL instances. There are three different sections used within a TASK declaration; PARAMETER, VARIABLE and SCHEDULE.

Parameter

The PARAMETER section is used to declare the actual arguments of a TASK. These are instantiated when the TASK is invoked.

Variable

The VARIABLE section is used to declare the local variables of a TASK. Theses variables are used in the operating schedule to hold temporary calculations.

Schedule

The SCHEDULE section is used to declare the operating schedule of a TASK. The operating schedule includes assignments and general equations defined to be applied on the actual arguments.

3.4 Process

A dynamic simulation experiment is defined using the PROCESS entity. A PROCESS contains several sections that is used to initiate and define the behavior of the simulation. For example, one can define solution parameters to the numeric solver and initiate model parameters. There are ten different sections in the PROCESS entity; UNIT, MONITOR, SET, EQUATION, ASSIGN, PRESET, SELECTOR, SOLUTIONPARAMETERS, INITIAL and SCHEDULE.

The UNIT, SET, EQUATION and SELECTOR sections are similar to those in a MODEL entity. The SCHEDULE section is similar to that of TASK.

Monitor

The MONITOR section is used to define which variables that are to be monitored. Variables left out will be suppressed.

Assign

The ASSIGN section is used to assign values to simulation variables. These are usually simulation specific.

Preset

The PRESET section is used to provide initial guesses for the variables in the simulation. This will override the default values for the variable types as defined in the DECLARE entity. These are usually related to the system being modeled.

Solutionparameters

The SOLUTIONPARAMETERS section is used to set numeric solution specific parameters.

Initial

The INITIAL section is used to define the initial state of the simulation.

Chapter 4

Analysis

gPROMS lacks a type system with type inference, so it is not possible to extend it with automatical dimensional analysis and inference without extending the language itself. Our approach is to develop a *semi polymorphic* dimensional type inference system for gPROMS. However, instead of calling our system a type inference system we will refer to it as a dimensional inference system, since it is not a general type system. Given this dimensional inference system, each symbol is associated with a dimension type (δ), representing its dimension. The dimension types are then used to infer dimensions and derive dimensional consistency.

The concept of semi-polymorphism is based on the idea of treating certain constructs as *polymorphic* and other as *monomorphic*. The polymorphism enables polymorphic parameterized models to be defined, while the monomorphism is required to safely analyze the dimensional consistency of the model equations. A polymorphic model can work on a range of different dimensions. One such example would be a polymorphic regulator model. The same general regulator could be applied to control a flow as well as a position. However, once the general model is instantiated we need the quantities to be monomorphic in order to safely check the dimensional consistency of the actual equations involved. Without the monomorphism certain dimensional inconsistencies would be left undetected, since that would allow different occurrences of the same symbol to have different dimensions.

4.1 Proposed Dimensional Inference System

Our dimensional inference system will be loosely based on the work of Wand and O'Keefe [WO91]. For instance, we will represent dimensions using rational exponents and use gaussian elimination for solving equations over dimensions. Likewise, our system will only handle dimensional analysis and inference of dimensions, and not automatic conversion of units. The extension to automatic unit conversion seems natural though and will be discussed in the future work section.

Basically, our system is based on transforming the equations that make up the simulation models into equivalent equations in the dimensional space. The generated equations specifies the dimensional constraints that result from dimensional analysis theorems. The constraints form a system of potentially independent linear equations, which could be solved using a number of different methods, for instance numeric algorithms, unification or gaussian elimination. We have chosen to solve them by means of Gauss-Jordan Row Reduction. The solutions to the equations are the inferred dimensions for the variables in order for the system to be dimensionally consistent. The dimensional analysis is performed at the same time, since a dimensionally inconsistent system does not have any solutions.

Our approach is only one of numerous possible, but one that seems sufficient and elegant. We do not want to restrict the expressiveness by using integer exponents, which is one of the issues discussed later. Also, we do not have to derive dimension types for functions, since the user is not allowed to define new functions in gPROMS. As a result we can use a simple gaussian elimination step for solving the equations.

4.1.1 Dimensional Information

In order for the dimensional analysis to work, the system needs information about the dimensions of the symbols that are used in the equations. Otherwise, the only solution we would find when solving the dimensional constraint equations would be the trivial solution — 0, which would indicate that each symbol is dimensionless.

gPROMS enables variable types to be accompanied by a unit of measure which would be possible to use as dimensional annotation. The drawback with this scheme is the requirement for unit conversions and the lack of dimensional information for the model parameters. Our solution is to extend gPROMS with the possibility to annotate variables and parameters with dimensional information.

4.1.2 Dependent Types

There is a problem regarding dependent types, types whose values are not statically known. For instance, one cannot know the value of a variable statically because it changes over time. This creates a problem when a dimension depends on a variable. One such case is expressions using the power operator. For example, consider the expression a^b. The dimension of the expression is dependent on the dimension of a and the value of b, but if b is a variable the dimension of the expression is undecidable. Previous work in dimensional inference for programming languages regard such expressions as dimensionless [Ken96, Rit95]. Since we do not agree with their view, our system has the ability to derive the correct dimensions for expressions using statically known exponents. However, there are some limitations to our approach — we can only handle rational exponents that are statically known (parameters and numeric literals). An extension to irrational exponents like $\sqrt{2}$ would require a fractional representation with loss of accuracy. So, whenever the exponent is statically unknown for the different reasons we have described, we flag a warning.

4.2 Special Language Elements of gPROMS

There are some issues that need to be worked out that relates to our approach and some of the basic language elements of gPROMS. For instance, how should we handle arrays; should we consider the possibility for each entry in an array to have a different dimension? How do we handle partial derivatives and integral constructions? How should we handle the intrinsic functions?

4.2.1 Models

A simulation is defined in terms of basic model entities that are constructed in a hierarchy of interconnected model instances, where each model is declared to be of a certain model type. A model declares certain parameters and variables which are used in the equation section of the model.

We will flatten the model hierarchy by expanding the equations and treat each variable and parameter instance separately. Also, it is important to create the actual model instances, because the dimensional consistency might depend on the value of the parameter instances.

4.2.2 Tasks

A function in gPROMS is defined in terms of a TASK, which describes an operating procedure using equations. Instead of deriving a polymorphic dimension type for each TASK, we will achieve polymorphism by treating each invocation of the task separately. This is accomplished by expanding the equations using the instantiated arguments and thus generating a new set of equations for each invocation relating to the specific arguments.

4.2.3 Arrays

gPROMS supports vector operations through an array construct. The models can contain array declarations of either parameters or variables. We have decided to restrict the use of arrays so that each entry in an array have the same dimension. This is assured by treating the whole array as a single variable or parameter.

4.2.4 Streams

As described earlier, streams are used to interconnect variables in different model instances. gPROMS basically expands the streams to equality relationships between the stream entries. We will do the same, asserting that each matching pair in the streams have equivalent dimensions.

4.2.5 Partial Derivatives and Integrals

In gPROMS it is possible to define partial derivatives and integrals when simulating distributed models. A variable can be declared as a distribution over one or more domains, and it is possible to derive or integrate the variable with respect to one or more of these domains. One can think of partial derivatives as division and integration as multiplication, although this is a great simplification.

Partial derivatives are defined using the `PARTIAL(expression, domains)` construct and turns out to be quite easy to handle. We will treat them much like division and divide the dimension of the expression by the dimension of the domain. We will not take in account how the expression is distributed. One could otherwise limit the use of partial by only allowing partial derivatives over the same domains that the expression is distributed over. Derivatives w.r.t time are treated differently and are defined with the special operator $.

Integrals are defined with the `INTEGRAL(ranges ; expression)` and are very general and can even be nested to arbitrary depth. This poses some problems which are related to the syntax and semantics of integrals in the gPROMS language. For example, consider the following mathematical expression and it's corresponding transformation to gPROMS:

$$\int_0^L z \cdot T(z)dz \Rightarrow \texttt{INTEGRAL(z:=0:L ; z*T(z))}$$

The resulting dimension is dependent on the local integration range z and the distributed variable T. The generality of integral expressions makes it somewhat complex to deduce the dimension of the result. For instance, what happens if the variable T is distributed over *Mass* and we are integrating with respect to *Length*. Is this physically sound? Moreover, should it be possible to infer dimensions through ranges and domains? Our view is that the dimension of the integral should depend on the distribution that is integrated and the local distribution domain declared in the integral expression. This design choice is subject to change if it proves to be inconsistent with the view of the model developers.

4.2.6 Foreign Objects

The foreign objects are external functions that are loaded during runtime when necessary. We need to know the dimensions of the input parameters and the dimension of the output. This information can be retrieved by communicating with the FO directly via the foreign object interface.

The foreign object interface supports a number of methods used to verify and extract information about a method in a given package. Two verification methods are gFOCM and gFOCMI, which verify the existence of a method in a package and return detailed information about its structure. For instance, one can retrieve conversion factors and the dimension of the input and output. There is one conversion factor for each input and output. The conversion factors are represented with an offset along with a multiplier. Therefore, gPROMS can convert, for instance, temperature measured in Fahrenheit to Celsius. The dimensions are represented with a vector of rational exponents of the fundamental or base dimensions. gPROMS uses the SI units extended with planar and solid angle along with dollars. Dollars is a required dimension for models that simulates economical dependencies, which is often an important factor in the industry.

The invocation of the special verification methods in the foreign objects to analyze them is not really in the scope of this thesis. The best suitable way to get the information would be to annotate the invocations of the foreign objects, which would be a cumbersome process. For now we will disregard the invocations of methods in foreign objects.

Chapter 5

A Dimension Type System

We will formalize our Dimension Type System using a small language, SMEL (Small Meta Equation Language), which is an implicitly typed monomorphic language that allows annotation of dimensions. SMEL is a simple language with support for general equations and declarations of variables with annotated dimensions. The reason why we are introducing this language, is because it will make the definition of our type system much more clear. Later we will show how the relevant constructions in gPROMS can be transformed into SMEL. However, there is a restriction on the transformation from gPROMS to SMEL. The underlying semantics of SMEL is not equivalent to that of gPROMS. For instance there is no notion of integrals in SMEL. It is only important that the transformation is dimensionally equivalent.

An EBNF grammar for SMEL is shown in Figure 5.1.

```
SMEL    = {eqtn} ;
eqtn    = expr, '=', expr ';'
        | id, '::', dim, ':=', (expr|"undef"), ';' ;
expr    = term, {('+'|'-'|'*'|'/'|'^'), term} ;
term    = ['-'], fact
fact    = '(', {eqtn}, expr, ')'
        | id, ['(', expr, ')']
        | num, '::', dim ;
dim     = '<', q, {',' q}, '>'
        | 0 (* zero vector *)
        |   (* unbound *) ;
q       = ['-'], num, ['/', num] ;
num     = {'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'}- ;
```

Figure 5.1: EBNF grammar for SMEL

As we can see from the grammar, the language is pretty small but powerful. For instance the language supports local declarations of variables within expressions, much like the **let** construction in functional languages. Also, we have added the capability to annotate the dimensions of variables and numeric constants. The language has been kept small by removing certain language properties usually found in normal programming languages. For instance, there is no precedence order for arithmetic operators. Precedence relationships are instead resolved by using compound expressions.

The ability to define equations within expressions is used to declare local variables and corresponding constraints on the values they can be assigned. For instance, `INTEGRAL` expressions in gPROMS define local domains. The domain dec-

larations are transformed into a variable declaration and a set of equations in the value domain, that restricts or specifies the values used for the domain.

5.1 Dimension Types

Each symbol is associated with a dimension variable ($\vec{d}$), which represents a vector of rational numbers, $\langle q_1, \ldots, q_n \rangle$, where n is the number of base dimensions. Each rational number corresponds to the exponent of a base dimension, determined by the position in the vector. A dimensionless quantity is represented with the zero vector ($\vec{0}$). Dimension types (δ) are built from linear combinations of dimension variables and constants closed under a vector space.

We will use the terms dimension types and dimensional expressions interchangeably. A dimension type is a dimensional expression and the generated constraints are thought of as equations relating two dimensional expressions.

The variables and parameters in gPROMS must not have polymorphic types, because our basic theorem states that dimensions are temporal invariant, which results in that each occurrence of a certain variable instance must have the same dimension. Therefore we have designed SMEL to be monomorphic. It turns out though, that the intrinsic functions must be given polymorphic types in order to work with all dimensions. Another exception is that gPROMS allows a limited kind of polymorphism by supporting parameterized models. This parameterization is easily handled during the transformation from gPROMS to SMEL.

SMEL implements the same intrinsic functions as gPROMS which include for example: exponential, trigonometric and scalar functions. The intrinsic functions are given polymorphic dimension types, represented by the type scheme: $\sigma_\delta = \forall \delta_1 . \delta_1 \rightarrow \delta_2$. According to the Π-theorem, the result of the exponential function must be dimensionless [WO91, Rit95]. The results of the trigonometric functions may also safely be assigned the dimensionless type. For example, the trigonometric functions are given the following type scheme:

$$\text{SIN} : \forall \delta . (\delta \rightarrow \vec{0})$$

Since our system is based on rational exponents, the square root function can be given the following type:

$$\text{SQRT} : \forall \delta . (\delta \rightarrow \frac{1}{2}\delta)$$

The scalar functions MIN, MAX, PRODUCT and SIGMA are all given the identity type. For example, the function SIGMA is used to sum all the elements in a vector. The dimension of the resulting scalar should be the same as the dimension of the vector. The identity type is defined in the following way:

$$\text{MIN} : \forall \delta . (\delta \rightarrow \delta)$$

The arithmetic operators are given the following types:

$$\begin{array}{rcl} +, - & : & \forall \delta . (\delta \times \delta \rightarrow \delta) \\ * & : & \forall \delta_1 \delta_2 . (\delta_1 \times \delta_2 \rightarrow \delta_1 + \delta_2) \\ / & : & \forall \delta_1 \delta_2 . (\delta_1 \times \delta_2 \rightarrow \delta_1 - \delta_2) \end{array}$$

It would be possible to assign a type scheme to each operator and treat them as ordinary functions. However, we will instead associate separate inference rules with each operator, since it is much more informative.

5.2 Dimension Type Rules

We now present a formal definition of the type rules that constitutes our dimension type system. Basically, the language elements of SMEL are associated with corresponding dimension type rules in our dimensional type system. Such type rules are often called judgments and are formalized in the following way:

$$\mathcal{C}, \Gamma \vdash e : \delta$$

In this judgment, $\mathcal{C}$ is a set of linear equations relating dimension type variables, Γ is a typing environment, e is an expression and δ is a dimension type. It reads "if identifiers are dimensionally constrained by $\mathcal{C}$ with respect to typing environment Γ, expression e has dimension δ". Each equation represents an equivalence relation between two dimension types, effectively forcing them to represent the same dimension. For instance, two quantities that are added must have the same dimension. The purpose of the type rules is to derive such a system of equations, constraining the dimensions and thus ensuring dimensional consistency for the entire simulation specification.

The typing environment (Γ) maps identifiers to their corresponding dimension variables ($\vec{d}$), and provides a context in which the dimensional inference takes place. Γ is a composition of two separate environments, $\Gamma_{\vec{d}}$ and Γ_{σ_δ}, according to $\Gamma = \Gamma_{\vec{d}} \cup \Gamma_{\sigma_\delta}$. $\Gamma_{\vec{d}}$ contains mappings between variables and their associated dimension variables, $[id \mapsto \vec{d}]$, and Γ_{σ_δ} contains mappings between functions and their polymorphic dimensions, $[id \mapsto \sigma_\delta]$. Bindings are looked up in Γ with the following semantics:

$$\mathcal{D}[\![id]\!]\Gamma = \begin{cases} \vec{d}, & id \text{ is a variable;} \\ \sigma_\delta, & id \text{ is a function;} \end{cases}$$

This implies that we cannot have variables with the same name as intrinsic functions, which is true for both gPROMS and SMEL.

Another environment, env_v, maps variables to their values. If an entry in env_v is bound, the value of the corresponding variable is statically known and can thus be used to infer the dimension of the exponent rule. A binding is looked up using a similar semantics as that for dimensions, $\mathcal{V}[\![symbol]\!]env_v = value$. If the variable is unbound, the lookup results in $undef$. The property of undef is the following:

$$undef + x = x + undef = undef$$

which is lifted to the obvious congruence over arithmetic operators.

We will also consider a function ϑ that takes as argument a general expression in SMEL and computes its value, using the environment env_v together with the usual semantics for arithmetic. The return values of function calls are *undef*.

Now we will present our typing judgments used for deriving the set of linear equations $\mathcal{C}$. They are as follows:

$$\text{eqtn; eqtn} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash \mathit{eqtn}_1 : \vec{0} \qquad \mathcal{C}_2, \Gamma \vdash \mathit{eqtn}_2 : \vec{0}}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash \mathit{eqtn}_1 \ ; \ \mathit{eqtn}_2 : \vec{0}}$$

$$\text{eqn} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash \mathit{expr}_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash \mathit{expr}_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\}, \Gamma \vdash \mathit{expr}_1 \ = \ \mathit{expr}_2 : \vec{0}}$$

$$\text{compound} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash \mathit{eqtns} : \vec{0} \qquad \mathcal{C}_2, \Gamma \vdash \mathit{expr} : \delta}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash \mathit{(eqtns \ ; \ expr)} : \delta}$$

$$\text{decl} \quad : \quad \frac{\mathit{env}_v \ = \ \mathit{env}_v \cup [\mathit{id} \mapsto q]}{\emptyset, \Gamma \cup [\mathit{id} \mapsto \vec{d}] \vdash \mathit{id}::\vec{d} := q : \vec{0}}$$

$$\text{add} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\}, \Gamma \vdash e_1 \ + \ e_2 : \delta_1}$$

$$\text{sub} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\}, \Gamma \vdash e_1 \ - \ e_2 : \delta_1}$$

$$\text{mul} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash e_1 \ * \ e_2 : \delta_1 + \delta_2}$$

$$\text{div} \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash e_1 \ / \ e_2 : \delta_1 - \delta_2}$$

$$\text{power}_1 \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \vec{0} \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash e_1 \uparrow e_2 : \vec{0}}$$

$$\text{power}_2 \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2 \qquad \vartheta(e_2) = 0}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash e_1 \uparrow e_2 : \vec{0}}$$

$$\text{power}_3 \quad : \quad \frac{\mathcal{C}_1, \Gamma \vdash e_1 : \delta_1 \qquad \mathcal{C}_2, \Gamma \vdash e_2 : \delta_2 \qquad \vartheta(e_2) \neq \mathit{undef}}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash e_1 \uparrow e_2 : \vartheta(e_2) \cdot \delta_1}$$

$$\text{app} \quad : \quad \frac{\mathcal{C}, \Gamma \vdash \mathit{expr} : \delta_1 \qquad \emptyset, \Gamma \vdash \mathit{id} : \delta_1 \rightarrow \delta_2}{\mathcal{C}, \Gamma \vdash \mathit{id} \ \mathit{(} \ \mathit{expr} \ \mathit{)} : \delta_2}$$

$$\text{spec} \quad : \quad \frac{\emptyset, \Gamma \vdash \mathit{id} : \sigma_{\delta_1}}{\emptyset, \Gamma \vdash \mathit{id} : \sigma_{\delta_1}[\delta/\delta_1]}$$

$$\text{id} \quad : \quad \frac{}{\emptyset, \Gamma \vdash \mathit{id} : \mathcal{D}[\![\mathit{id}]\!]\Gamma}$$

5.3 Transformation to SMEL

Now we will show how gPROMS can be transformed into SMEL. We will give an informal description of the transformation for the different language elements of gPROMS, as well as define the transformation rules. A transformation rule maps a syntactic construction in gPROMS to a *dimensionally equivalent* syntactic construction in SMEL. Parts of the gPROMS language that are not dependent on dimensional consistency are skipped during the transformation simply by not defining the corresponding transformation rules. Basically, after the transformation we are left with the equations in the simulation models and the symbols that occur in them. The symbols are either parameters or variables in gPROMS, which are treated the same in SMEL.

5.3.1 Polymorphic Parameterization

During the transformation, the polymorphic parameterization must be taken care of. This is asserted by treating each instance separately. The symbols in the resulting dimensional equations are based on instance specific information. For example, if we have a parameter b in MODEL a we refer to this symbol as $a.b$. This could be viewed as creating an instance of a type scheme for that symbol, which resembles how a polymorphic type system would be constructed. However, our approach limits the polymorphism so that for a given symbol instance it will at most be bound to one specific dimension.

5.3.2 Dimensional Information

Each symbol in gPROMS can either be associated with an annotated dimension, or left unbound. Unbound symbols can later be bound to what ever dimensions are necessary to uphold dimensional consistency. The limitation is that due to the temporal invariance, they can only be bound once. Therefore it would be wrong to say that the symbols are polymorphic.

The annotations in gPROMS are transformed into equivalent annotations in SMEL, by annotating the corresponding declarations. It is important to notice that there are no parameters in SMEL, only variables. For instance if a parameter a in model b is annotated to m/s, the corresponding variable in SMEL, $a.b$, is also annotated to m/s. The same applies for variables in gPROMS, only now the annotated dimension for the specific variable type is used. For instance if a variable type *Meter* in gPROMS is annotated to m, all instances of type *Meter* are also annotated to m. If a variable type is unbound, all variable instances will also be unbound, which ensures the possibility to create polymorphic parameterized models.

Numeric Constants

Numeric constants are explicitly typed during the transformation depending on the context. If a numeric constant is part of a multiplicative expression it must be dimensionless, but if it is part of an additive expressions it must be polymorphic. The effect of this scheme is that the numeric constants cannot be used to infer dimensions or balance formulas. We believe that it is more safe to infer dimensions for statically known quantities through parameters.

In practice, each physical constant that is used should be defined as a parameter in gPROMS. In this way it is assured that the constant is used dimensionally consistently.

5.3.3 Parameter Values

In gPROMS, parameters can be set to constant values statically before the simulation begins. These value assignments are moved to the declarations of the parameters. For instance, consider a dimensionless parameter b in model a ($a.b$) which is statically assigned a value of 100. The corresponding declaration of $a.b$ in SMEL would be:

```
a.b::0 := 100;
```

In the same way, we must prevent that variables are assigned values. Therefore all variables are declared as `undef`, like follows:

`variable::`*dimension* `:= undef;`

5.3.4 Transformation Rules

We will now formalize the transformation from gPROMS to SMEL by defining the transformation rules. As described earlier, the transformation rules are used to map dimensionally equivalent syntactic constructs in gPROMS to SMEL. We will define the rules by using a mapping function, f, which ranges over the different syntactical constructs of gPROMS. We will only define the output mappings of f for the mathematical operations of gPROMS, as the other constructions are trivial and most of them are not needed anyway.

Mathematical Operators

As previously mentioned we need to treat the numeric constants differently depending on the context. They are explicitly typed to dimensionless in multiplicative constructs (`::0`) and explicitly typed to polymorphic in additive constructs using the special polymorphic typing (`::`). Since the arithmetic operators are associative we will not specify each possible input combination.

$$
\begin{array}{lcl}
f(\texttt{expr (<|>|<=|>=|=) expr}) & = & \texttt{expr = expr} \\
f(\texttt{expr (and|or) expr}) & = & \texttt{expr = expr} \\
f(\texttt{expr (*|/|+|-) expr}) & = & \texttt{(expr (*|/|+|-) expr)} \\
f(\texttt{num (*|/) expr}) & = & \texttt{(num::0 (*|/) expr)} \\
f(\texttt{num (+|-) expr}) & = & \texttt{(num:: (+|-) expr)} \\
f(\texttt{expr \^{} expr}) & = & \texttt{(expr \^{} expr)}
\end{array}
$$

Derivatives

The special derivative with respect to time ($) is handled by dividing the expression with a numeric constant of dimension $Time$ ($\langle 0,0,1 \rangle$). The partial derivative is simply treated as a division.

$$
\begin{array}{lcl}
f(\texttt{\$ expr}) & = & \texttt{(expr / 1::<0,0,1>)} \\
f(\texttt{PARTIAL(expr,domain)}) & = & \texttt{(expr / domain)}
\end{array}
$$

Integrals

Integrals are treated much like normal multiplication with the addition of local domain declarations. These declarations are translated to additional equations within the integral expression. In case of multidimensional integrals one or more domains might be declared, which is not a problem during the translation.

$$
f(\texttt{INTEGRAL(dom:=e1:e2;expr)}) \quad = \quad \texttt{(dom:: :=e1;dom:: :=e2;expr*dom)}
$$

5.4 Dimensional Inference Algorithm

We will now give a description of our dimensional inference system. Later in this chapter we will give a more thorough explanation of the algorithms involved at each step. These are the basic steps of our inference engine:

1. Derive system of equations, $\mathcal{C}$.
2. Infer trivial dimensions.
3. Solve systems of equations.

5.4.1 Overview

Before we go into the details about our algorithms, we will briefly describe each of the involved steps.

Derive System of Equations

First, the system of equations $\mathcal{C}$ is derived using the type rules described earlier. In practice, the transformed gPROMS source specification is parsed and converted into an abstract syntax tree (AST) whereafter the type rules are applied recursively on the structure. The resulting system of equations contains all dependencies necessary to analyze the dimensional consistency.

Infer Trivial Dimensions

Before we solve the system of equations we use the statically annotated information to recursively infer new dimensions, by an extended back substitution algorithm. The process is based on finding equations with only one unknown, which are trivially solved. When such an equation is found, the binding for the unknown dimension variable is updated and all occurrences of it are substituted for its dimension. Any resulting redundant equations are removed. This part of the algorithm terminates when there are no more trivial equations to solve.

During this step it is possible to detect dimensional inconsistencies simply by finding equations where the two sides are not dimensionally equivalent. Such a property is easily checked once all dimension variables in an equation are known.

One of the main reasons why we introduce this step is because we believe that given enough dimensional annotations, this step will reduce the overall execution time and at the same time offer qualitative dimensional inconsistency reports. The dimensional inconsistencies found during Gauss-Jordan Row Reduction are hard to trace, since the structure of the system is destroyed during Row Reduction.

Solve Systems of Equations

If not all equations were removed in the previous step, the system of equations is divided into one or more partitions which can be solved independently. A partition is defined as the least system of equations satisfying the condition that each dimension variable must only occur in the equations of one specific partition.

The partitioned systems of equations are finally solved using Gauss-Jordan Row Reduction. If the systems prove to be uniquely solvable, we end up with the inferred dimensions for the unknown variables. Otherwise we have either no solutions or infinite number of solutions. In case of no solutions we have dimensional inconsistency, and in case of infinite number of solutions the system is consistent, but there are still unknown dimensions. We do not experience any of the numerical problems usually attributed to Gaussian elimination since we have based our computations on rational numbers.

5.4.2 Infer Trivial Dimensions

By using the annotated dimensions, we can statically derive new dimensions by means of *equational unification* and *substitution*. Depending on the amount of known dimensions and how the equations interact the complexity of the final inference step by Gaussian elimination can be greatly reduced. In fact, if enough dimensions are annotated we can solve the entire system by means of recursive substitution and unification.

Representation

The dimensional constraint equations are represented as a 3-tuple $\langle \mathcal{T}, \vec{d}_{\text{SET}}, \vec{q} \rangle$, where $\mathcal{T}$ is a list of terms, $\vec{d}_{\text{SET}}$ is a set of the occurring dimension variables and $\vec{q}$ is the accumulated result of the substituted dimensions. Each term is represented as a 2-tuple, $\langle q, \vec{d} \rangle$, where q is a rational factor and $\vec{d}$ is a dimension variable. For example, the equation $2\vec{d}_a = \vec{d}_b$ is represented as $\langle [\langle 2, \vec{d}_a \rangle, \langle -1, \vec{d}_b \rangle], \{\vec{d}_a, \vec{d}_b\}, \vec{0} \rangle$, since $2\vec{d}_a = \vec{d}_b \Leftrightarrow 2\vec{d}_a - \vec{d}_b = \vec{0}$.

The dimension variables are usually represented by their identifiers, but they also have other information associated with them. The actual representation of a dimension variable is a 3-tuple $\langle id, \vec{q}, \mathit{known} \rangle$, where id is the identifier, $\vec{q}$ is the dimension it represents and *known* is a boolean flag indicating whether the dimension is known or not.

Later, as we describe our algorithms, we will need to refer to the individual elements of these compound data types. For instance, if t is a term then $\vec{d}[t]$ is its dimension variable, and if v is a dimension variable then $\vec{q}[v]$ is its dimension.

Algorithms

We have developed two algorithms that are invoked in sequence on the set of equations. The first algorithm will simplify the equations so that each symbol in an equation only occurs once. The second algorithm will recursively substitute terms in the equations for known dimensions. After these two algorithms have been applied, we are left with a possibly reduced set of equations containing only unknown variables, depending on the amount of known dimensions.

Simplifying the Equations

First we are going to reduce the equations using Algorithm 5.1. This algorithm works by iterating through the terms of each equation and merging terms representing the same symbol, thereby forcing each symbol to only occur once in an equation.

Algorithm 5.1: Simplify Equations

```
SimplifyEquations(Es)
  input  : Es, the set of equations to be reduced
  output : Es, the set of reduced equations

  begin
1   foreach equation e_i ∈ Es do
2     foreach term t_j ∈ T[e_i] do
3       foreach term t_k ∈ T[e_i] ∧ t_k ≠ t_j do
4         if d[t_j] = d[t_k] then
5           q[t_j] ← q[t_j] + q[t_k]
6           T[e_i] ← T[e_i] − {t_k}
7       if q[t_j] = 0 then
8         T[e_i] ← T[e_i] − {t_j}
9         d_SET[e_i] ← d_SET[e_i] − {d[t_j]}
10    if |T[e_i]| = 0 then
11      Es ← Es − {e_i}
  end
```

By using simple algebraic techniques it is also possible to remove equations that are completely redundant. For example, consider the following equation:

$$\begin{aligned} 2\vec{d}_a &= \vec{d}_a + \vec{d}_a \quad \Rightarrow \\ 2\vec{d}_a - \vec{d}_a - \vec{d}_a &= \vec{0} \end{aligned}$$

It is easy to realize that this equation is redundant independently of the dimension of a, since the terms will cancel each other out and we are left with the following equation:

$$0\vec{d}_a = \vec{0}$$

Clearly this equation does not add any information and can safely be discarded. We do not have to check that the dimension really is dimensionless, since we have not expanded any statically known dimensions yet.

Recursive Substitution of Dimensions

When the equations are reduced we can apply Algorithm 5.2 in order to expand the statically known dimensions.

Algorithm 5.2: Recursive Substitution of Dimensions

```
RecursiveSubstitution(Es, E)
  input  : Es, the set of all equations
  input  : E, the set of equations to process
  output : Es, the set of resulting equations

  begin
1   foreach equation e_i ∈ E do
      /* Substitute for known dimensions */
2     foreach term t_j ∈ T[e_i] do
3       if known[d[t_j]] = true then
4         q[e_i] ← q[e_i] − q[t_j] * q[d[t_j]]
5         T[e_i] ← T[e_i] − {t_j}
      /* If there are no terms left in equation — remove it */
6     if |T[e_i]| = 0 then
7       Es ← Es − {e_i}
8       E ← E − {e_i}
9       if q[e_i] ≠ 0 then
10        ReportError("Dimensional inconsistency")
      /* If there is exactly one term in equation — unify dimension */
11    if |T[e_i]| = 1 then
12      t ← t_1 ∈ T[e_i]
13      q[d[t]] ← q[e_i]/q[t]
14      known[d[t]] ← true
15      Es ← Es − {e_i}
16      E ← E − {e_i}
        /* Build the set of equations that are dependent on this symbol and
        solve the variables in these equations recursively */
17      NewSet ← {e : (e ∈ Es ∧ d[t] ∈ d_set[e])}
18      RecursiveSubstitution(Es, NewSet)
  end
```

Algorithm 5.2 recursively substitutes variables for their known dimensions until there are no more dimensions to substitute. New dimensions are derived by finding equations with only one unknown and then trivially solve for its dimension. The equation is then removed from the set of equations, since there is no further information in it. The algorithm then recurses over the equations that are dependent on

the previously unknown variable. This whole process is executed iteratively until there are no more dimensions to substitute.

For example, consider the following two dimensional equations:

$$\begin{aligned} \vec{d_a} &= \langle 0, 1, -2 \rangle \\ 2\vec{d_b} - \vec{d_a} &= \langle 0, 0, 0 \rangle \end{aligned}$$

The two variables, $\vec{d_a}$ and $\vec{d_b}$ are unknown, and thus placed to the left. The algorithm will find the first equation and trivially solve for the dimension of $\vec{d_a}$, which is $\langle 0, 1, -2 \rangle$. Now, each occurrence of $\vec{d_a}$ is substituted for $\vec{q}[\vec{d_a}]$ and the first equation is removed, resulting in the following equation:

$$2\vec{d_b} = \langle 0, 1, -2 \rangle$$

This equation is trivially solved by the algorithm and the solution is the dimension of $\vec{d_b}$, $\langle 0, \frac{1}{2}, -1 \rangle$. The equation is removed and the algorithm terminates since there are no more equations. In this case, the system was proved to be consistent and complete.

However, what if the dimension of $\vec{d_b}$ is also known in the equation above. For example, consider the case where the dimension of $\vec{d_b}$ is equal to $\langle 0, 0, -1 \rangle$. When the dimension is substituted in the equation above, we end up with the following equation:

$$\langle 0, 0, 0 \rangle = \langle 0, 1, 0 \rangle$$

Clearly we now have a dimensional inconsistency, since the sum of all the dimensions must be dimensionless. So in this case the system was proved to be inconsistent. Also, the inconsistency can be traced to this particular equation. This is not quite right though, but will give some pointer to where things might be wrong.

5.4.3 Solve Systems of Equations

In order to infer the missing dimensions, we must solve the system of equations for the unknowns. The system of equations is solved using Gauss-Jordan Row Reduction method, since it is easily computerized and relatively efficient.

There are three possible outcomes, a single solution, an infinite number of solutions or no solutions. In case of no solutions, we have an inconsistent system, which can never be satisfied. If we have exactly one solution, the system is solved and each previously unknown variable is now known with a most general dimension. If we have infinite number of solutions, we have a consistent system with dependencies between the unknown variables. The case of infinite solutions will be explained in more detail later.

Partition Equations

Before we apply Gauss-Jordan Row Reduction, the equations are partitioned into several independent systems of equations using Algorithm 5.3. It is important to partition the equations since the complexity of Gauss-Jordan Row Reduction is heavily dependent on the size of the system of equations. Thus, we can minimize the the total amount of time necessary to solve the equations by minimizing the size of each independent system. Also, the partitioning enables us to trace the dimensional errors to their origin (all equations in the partition).

Algorithm 5.3: Partition Equations

```
PartitionEquations(Es)
  input  : Es, the set of equations to be partitioned
  output : Ps, the set of resulting partitions

  begin
1   OldList ← ∅
2   while |Es| > 0 do
      /* Pick some arbitrary equation e in Es */
3     e ← e ∈ Es
4     Es ← Es − {e}
5     P ← {e}
6     WorkList ← {s : (s ∈ d⃗_SET[e])}
      /* Build partition originating from equation e */
7     while |WorkList| > 0 do
8       symbol ← s ∈ WorkList
9       WorkList ← WorkList − {symbol}
10      OldList ← OldList ∪ {symbol}
        /* Add equations that symbol occurs in to partition */
11      foreach e_i ∈ Es do
12        if symbol ∈ d⃗_SET[e_i] then
13          P ← P ∪ {e_i}
14          Es ← Es − {e_i}
15          WorkList ← WorkList ∪ {s : (s ∈ d⃗_SET[e_i] ∧ s ∉ OldList)}
16    Ps ← Ps ∪ {P}
  end
```

Gauss-Jordan Row Reduction

The Gauss-Jordan Row Reduction method is a highly systematic algorithm for solving systems of linear equations. The equations are organized in an augmented matrix, where each row corresponds to an equation. The basic steps of the algorithm is the row operations. These operations are performed on the augmented matrix for a linear system until it is transformed into a new augmented matrix, called reduced row echelon form. The reduced row echelon form reveals the solutions without any need for back substitution, which is otherwise required for normal Gaussian elimination.

The following three operations are legal operations on the rows:

1. Multiplying a row by a nonzero scalar.

2. Adding a scalar multiple of one row to another row.

3. Switching the positions of two rows in the matrix.

When simplifying the augmented matrix associated with a system, we work on one column at a time. Initially we begin with the first nonzero column of the matrix. After a column is completely simplified, we move to the next column to the right. While working on a particular column, one row is singled out as special. This is called the *homerow*. We begin with the first row as homerow and move down as the reduction progresses. The matrix entry that lies in both the current homerow and in the current column is called the *pivot*. The algorithm terminates either when we run out of rows to use as the homerow, or run out of columns before the augmentation bar. See Algorithm 5.4.

Algorithm 5.4: Gauss-Jordan Row Reduction

```
GaussJordanRowReduction(M)
  input  : M, a matrix representing the system of equations
  output : M, the resulting matrix in row echelon form

  begin
1   homerow ← 0
2   i ← 0
3   n ← rows[M]
4   m ← cols[M]
5   while homerow < n ∧ i < m do
6     Find pivot p, such that (M_{p,i} > 0 ∧ homerow ≤ p < n)
7     if p found then
8       Switch M_{homerow} with M_p
9       for c ← i to m do
10        Multiply M_{homerow,c} with 1/M_{homerow,i}
11      foreach row r ∈ M ∧ r ≠ homerow do
12        for c = i to m do
13          Add −M_{r,i} * M_{homerow,c} to M_{r,c}
14      homerow ← homerow + 1
15    i ← i + 1
  end
```

Infinite Number of Solutions

After row reduction, columns with nonzero pivot elements are often labeled as pivot columns, and the others are called non pivot columns. The variables for non pivot columns are called independent variables, and the others are called dependent variables. If a given system is consistent, solutions are found by letting each independent variable take on any real value (or rational value). The values of the dependent variables can then be computed from these choices.

In our case, for the system to be complete it needs information about these independent variables. This information can be entered in form of annotations. An important factor is what variables that should be annotated, and it is our system that must suggest these variables. This requires some form of heuristics. We want the user to annotate the variables that affects the system the most, i.e the ones that occur most frequently. However, these variables must be chosen among the independent, since these control the value of the dependent ones. Therefore we must come up with a heuristic that favors the variables that occur the most to become independent variables. A simple heuristic that works in most cases is to sort the variables according to the number of equations that they appear in, so that the ones that occur the most are placed to the right. This works, because the Gauss-Jordan Row Reduction works its way from left to right. It is more likely for a variable that is chosen first to become a dependent variable than one that is selected last.

Chapter 6

Implementation

We have implemented our dimensional inference algorithms for gPROMS using Java. The implementation is in the form of a stand-alone tool that statically analyzes the dimensional consistency of a gPROMS simulation model. We decided to omit certain language features due to mostly time constraints. The limitations will be discussed further in this chapter.

The algorithms are implemented slightly different from the pseudo-code shown previously, since we have used an object oriented approach. Also, our representation of the interconnections between the equations and the variables form a dependence graph. The use of a dependence graph removed some of the iterations, reducing the overall computational complexity of the algorithms.

6.1 Overview

First the gPROMS source program is parsed and converted into an intermediate representation of SMEL in the form of an AST. During this transformation redundant language constructions are skipped. Also, in order to simplify derivation of certain dimensions the static expressions are evaluated and reduced to numeric literals.

The dimensional constraint equations are derived by traversing the AST and applying the type rules of our dimension type system to each node. Since much of the work is done during the transformation, such as flattening the model hierarchy, the derivation of the system of constraint equations is fairly easy. When each node is traversed, the dimensional consistency is analyzed using the algorithms described previously.

Our tool also provides detailed descriptions of the dimensional errors that are found and directs the user into the specific lines in the source where the error is located. The system also provides the heuristics for deciding which parameters or variables that should be annotated. This will most likely be of great use for the users in order to localize errors in their source.

6.2 Syntax Extensions

The original gPROMS syntax was altered in a few ways, which we will describe here. First off, it was made more similar to normal programming languages without affecting compatibility with the gPROMS environment. We also added the annotation capability that is needed for the dimensions.

6.2.1 Homogenization of gPROMS Syntax

In order to simplify the analysis, the original gPROMS syntax has been altered to make the syntax more homogeneous. Some constructions have been merged together in order to minimize the number of special cases that otherwise would need to be handled. At the same time the syntax has been made more similar to a normal imperative programming language.

The new syntax is a proper superset of gPROMS, so it can process a larger set of input data than the original syntax. The downfall is that it is possible for the system to process input data that is not a proper gPROMS file. This is not such a large problem however, since we require that each input must be a proper gPROMS input file.

6.2.2 Dimensional Annotations

We have decided to use the SI system of units as acronyms for the actual physical dimensions. This means that instead of using, for instance, the dimension *Mass*, the SI unit *kilogram* is used instead. The reason for this convention is because gPROMS uses SI units internally when communicating with the foreign objects, and the fact that the SI system of units is standardized and commonly used in the industry. We have chosen to use the 7 standard units, the 22 derived units and dollars.

Also, we do not use the unit specification to annotate dimensions, since the model developers might want to use it for their own unit of measures. Instead, we have extended the gPROMS language with annotation of dimensions for both variable types and parameters. These annotations are placed within comments so the source program will still be compatible with the gPROMS environment. There are two types of comments in gPROMS, line comment and block comment. We have chosen to use the block comment `{...}` for dimensional annotation. The special annotation comment looks like this: `{@dim ... }`, with ... replaced by the actual dimension.

```
dimAnnot   =  "{@dim" dimEntity, {('/' | '.'), dimEntity} '}'
dimEntity  =  ident, ['^', rational]
rational   =  ['-'], integer, ['/', integer]
```

Figure 6.1: EBNF grammar for dimensional annotations

An important issue regarding the annotations is that they must be intuitive, to make it easier to annotate the symbols. Otherwise, model developers might skip using the tool at all. To accommodate this a simple syntax was developed, where complex derived units can be expressed with ease. For instance the derived SI unit N can be annotated in the following way: `{@dim kg.m/s^2}`. An EBNF grammar describing the syntax is shown in figure 6.1.

```
{@dim m/s^2}
{@dim m.s^-2}
{@dim m/s.s}
{@dim m/s/s}
```

Figure 6.2: A few different ways to annotate acceleration

The operators `'-'` and `'/'` are both right associative in order to implement the correct semantics for annotations like the following: `{@dim m/s/s}`, which shows

one way to annotate acceleration. Basically both operators are forms of unary negation where '/' is used to negate a compound dimensional entity, whereas '-' is used to negate the rational exponent of a simple dimension. See Figure 6.2 for a list of some of the different ways one can annotate the dimension for acceleration in.

6.3 Limitations

Our system does not handle every construction in the gPROMS language. For instance, we decided to skip the foreign objects, since it is just a matter of software engineering and could easily be added later. Also, we skipped MODEL inheritance capabilities. We are only considering the variables and parameters explicitly declared within a MODEL. This feature could also be added later.

Chapter 7

Results

In this chapter we will show some simple models and the corresponding result after running them through the dimensional analysis tool, which will enlighten some of the capabilities of our system. The models are small fictitious models constructed solely to serve the above mentioned purpose. Although the models are fictitious they are built upon physical formulas commonly used in engineering. The chapter will be divided into a few different scenarios, each scenario enlightening some specific feature or features of our system. Also, see [DPL03] for an example of a buffered tank model.

```
1   DECLARE
2     TYPE
3       Current    = 1.0 : -1E10 : 1E10 {@dim A}
4       Voltage    = 1.0 : -1E10 : 1E10 {@dim V}
5       Resistance = 1.0 : -1E10 : 1E10 {@dim ohm}
6   END
7
8   MODEL Test1
9     VARIABLE
10      I AS Current
11      U AS Voltage
12      R AS Resistance
13    EQUATION
14      U = R * I;
15  END
16
17  PROCESS Simulate
18    UNIT
19      Test AS Test1
20  END
```

Figure 7.1: Source specification for Test1

7.1 Dimensional Analysis

One of the main features of our system is the capability to statically analyze the dimensional homogeneity of a simulation model. A pure dimensional analysis check requires that all dependent symbols occurring in the model equations are annotated

with the proper dimensions.

In this scenario the model specified in Figure 7.1 will be used, which makes use of ohms' law, describing the relation between electric current, voltage and resistance. As we can see, each symbol is associated with an annotated dimension, which makes it possible to determine the dimensional consistency of the model, using only dimensional analysis. The following result was reported from our system:

```
Parsed DECLARE
Parsed MODEL    TEST1
Parsed PROCESS  SIMULATE

System is consistent and complete.
No warnings.
```

As we can see, the system had no difficulty deducing the dimensional consistency, given that all the symbols were annotated.

7.2 Dimensional Inference

We have seen that the system is able to handle models in which all symbols are properly annotated. But, what happens if we remove the dimensional annotation of `Voltage` in the previous scenario, yielding the model according to Figure 7.2. The following output was reported:

```
Parsed DECLARE
Parsed MODEL    TEST2
Parsed PROCESS  SIMULATE

Inferred Dimensions:
TEST.U = V = <m2.kg.s-3.A-1>

System is consistent and complete.
No warnings.
```

The output is not so surprising considering we have one equation with only one unknown, which is trivially solved. The system infers the dimension of the Voltage symbol during the "application of known dimensions" step.

If we were to remove yet another annotation in the scenario above, that for `Current`, we end up with the following output:

```
Parsed DECLARE
Parsed MODEL    TEST2
Parsed PROCESS  SIMULATE

Unknown Symbol(s):
CURRENT @ 3
VOLTAGE @ 4

Symbol(s) To Annotate:
CURRENT @ 3

System is consistent but not complete.
No warnings.
```

```
1   DECLARE
2     TYPE
3       Current    = 1.0 : -1E10 : 1E10 {@dim A}
4       Voltage    = 1.0 : -1E10 : 1E10
5       Resistance = 1.0 : -1E10 : 1E10 {@dim ohm}
6   END
7
8   MODEL Test2
9     VARIABLE
10      I AS Current
11      U AS Voltage
12      R AS Resistance
13    EQUATION
14      U = R * I;
15  END
16
17  PROCESS Simulate
18    UNIT
19      Test AS Test2
20  END
```

Figure 7.2: Source specification for Test2

This is not so surprising either, since we now have only one equation but two unknowns, which has infinite number of solutions. However, the system is still consistent though, and the dimensions of the two symbols `Current` and `Voltage` can be described as a linear combination in terms of other symbols. Instead of expressing the relationship between the dependent symbols, the system heuristically proposes that the symbol `Current` should be annotated.

We know that annotating the symbol `Current` would solve the problem, since that is exactly the case above. But, what if we add a new equation also relating the unknown symbols `Current` and `Voltage`, so we end up with two equations and two unknowns? We added an equation relating electric power as the product of voltage and current, $Power = Voltage * Current$. Since we do not have the symbol Power we also added this. The altered model is specified in Figure 7.3. The following output was generated:

```
Parsed DECLARE
Parsed MODEL    TEST3
Parsed PROCESS  SIMULATE

Inferred Dimensions:
CURRENT = A = <A>
VOLTAGE = V = <m2.kg.s-3.A-1>

System is consistent and complete.
No warnings.
```

Since `Power` and `Resistance` are known, we end up with two equations and two unknowns which forms a system of equations solvable by means of linear algebra. The solution reveals the dimensions for the unknown symbols, `Current` and `Voltage`.

```
1   DECLARE
2     TYPE
3       Current    = 1.0 : -1E10 : 1E10
4       Voltage    = 1.0 : -1E10 : 1E10
5       Resistance = 1.0 : -1E10 : 1E10 {@dim ohm}
6       Power      = 1.0 : -1E10 : 1E10 {@dim W}
7   END
8
9   MODEL Test3
10     VARIABLE
11      I AS Current
12      U AS Voltage
13      R AS Resistance
14      P AS Power
15    EQUATION
16      U = R * I;
17      P = U * I;
18  END
19
20  PROCESS Simulate
21    UNIT
22      Test AS Test3
23  END
```

Figure 7.3: Source specification for Test3

7.3 Dimensional Inconsistency

In this scenario we will demonstrate what happens if a simulation model is not dimensionally consistent. The user will get some warnings indicating what kind of conflict or inconsistency that occurred and where in the source specification it is located. Dimensional inconsistencies are best detected if as many symbols as possible are annotated, otherwise we might just end up inferring erroneous dimensions.

We will consider a simple simulation model containing dimensional inconsistencies and take appropriate actions based on the results from our system. The model we will be using is specified in Figure 7.4. The following output was reported:

```
Parsed DECLARE
Parsed MODEL    TEST4
Parsed PROCESS  SIMULATE

WARNING @ 23: Inconsistent equation. dimension = <m>
WARNING @ 25: Inconsistent equation. dimension = <s-1>

System is inconsistent.
2 warnings.
```

The two warnings we see correspond to dimensional inconsistencies in the simulation model. At line 23 we have the equation `P = F / d` which is supposed to compute the pressure `P`. The dimension specification in the warning informs of the dimensional difference between the two sides of the equation, in this case `<m>`. In this case it is easy to realize that we have made a typing error, since Pressure is defined in terms of force per square length. Therefore we adjust the equation at

```
1   DECLARE
2     TYPE
3       Velocity     = 1.0 : -1E10 : 1E10 {@dim m/s}
4       Length       = 1.0 : -1E10 : 1E10 {@dim m}
5       Mass         = 1.0 : -1E10 : 1E10 {@dim kg}
6       Acceleration = 1.0 : -1E10 : 1E10 {@dim m/s^2}
7       Work         = 1.0 : -1E10 : 1E10 {@dim N.m}
8       Force        = 1.0 : -1E10 : 1E10 {@dim N}
9       Pressure     = 1.0 : -1E10 : 1E10 {@dim Pa}
10  END
11
12  MODEL Test4
13    VARIABLE
14      m AS Mass
15      a AS Acceleration
16      d AS Length
17      W AS Work
18      F AS Force
19      P AS Pressure
20      v AS Velocity
21    EQUATION
22      F = m * a;
23      P = F / d;
24      W = F * d;
25      v = a;
26  END
27
28  PROCESS Simulate
29    UNIT
30      Test AS Test4
31  END
```

Figure 7.4: Source specification for Test4

line 23 to the following: `P = F / d^2`, adjusting for the missing `<m>`. Similarly it is easily discovered that the equation at line 25 is erroneous, since velocity does not equal acceleration. Instead, acceleration is defined as the time derivative of velocity, indicated by the dimension `<s-1>` in the warning. Therefore we alter the equation to the following: `$v = a`. When we run the simulation model again we end up with the following output:

```
Parsed DECLARE
Parsed MODEL    TEST4
Parsed PROCESS  SIMULATE

System is consistent and complete.
No warnings.
```

Clearly, guided with the warnings it was possible to locate the dimensional inconsistencies and resolve them. Given more complex simulation models it might not be this easy to locate the actual source of a possible dimensional inconsistency, but the warnings will point in the right direction.

Chapter 8

Conclusions and Future Work

In this thesis we have shown a dimensional inference system for gPROMS and motivated its usefulness in the industry. Our proposed system has been implemented as a stand-alone tool in Java, which is to be used for evaluation of dimensional analysis and inference in modeling languages. So, it is only when the model developers have been using our tool for some time that we can actually draw any conclusions on how useful our tool really is. If the system proves to be valuable in model developing, a future extension might be to incorporate it into the gPROMS environment.

However, there are some pieces missing in our system which might be added later on. These include the use of foreign objects and model inheritance. Also, the way the dimensions are annotated might also be changed along with the internal representation of dimensions. The user might want to use another set of base dimensions than the ones defined in SI.

It would also be interesting to extend our system with the capabilities of automatic unit conversion. This could be added by extending our data structures to hold scaling information for each unit to the corresponding SI unit. However, if the system should be able to handle conversions between other than commensurate units one need to generalize the conversion factors into conversion functions. These functions could be annotated in a simple syntax along with the unit of measure.

Appendix A

EBNF Grammar for gPROMS

```
(* 2002-04-10 Daniel Persson *)

gPROMS = {
    declareBlock
  | modelEntity
  | taskEntity
  | processEntity } ;

(* - DECLARE entity ------------------------------------- *)
declareBlock = "declare", {declareSection}, "end" ;
declareSection = ("type", {variableTypeDecl})
  | ("stream", {streamTypeDecl}) ;
variableTypeDecl = ident, variableInitiation, [unitAnnot] ;
variableInitiation =
    '=', expression, ':', expression, ':', expression ;
unitAnnot = "unit", '=', strLiteral ;
streamTypeDecl = ident, "is", identifierList ;

(* - MODEL entity --------------------------------------- *)
modelEntity = "model", ident, ["inherits", ident]
  , {modelElementDecls}, {setSection}, {boundarySection}
  , {equationSection}, "end" ;
modelElementDecls = parameterSection
  | distributionDomainSection
  | unitSection
  | variableSection
  | streamSection
  | selectorSection ;

(* Parameters *)
parameterSection = "parameter", {parameterDecl}- ;
parameterDecl = identifierList, "as", parameterType ;

parameterType = basicParameterType
  | arraySpec, basicParameterType
  | foreignObjectParameterType ;

basicParameterType = basicType, [defaultValue] ;
defaultValue = "default", expression ;
```

```
foreignObjectParameterType =
    "foreign_object", [strLiteral], ["default", strLiteral] ;

(* Distribution_domains *)
distributionDomainSection =
    "distribution_domain", {domainDecl}- ;
domainDecl = identifierList, "as", domainType ;
domainType = [arraySpec], basicDomain ;
basicDomain = ('(', domainExpression, ')'
  | '[', domainExpression, ']') ;
elementDistributionSpec = : arraySpec | distributionSpec ;
arraySpec = "array", '(', distributionList, ')', "of" ;
distributionSpec =
    "distribution", '(', distributionList, ')', "of" ;
distributionList = distributionEntity
  , {',', distributionEntity} ;
distributionEntity = domainExpression | expression ;

(* Units *)
unitSection = "unit", {unitDecl}- ;
unitDecl =
    identifierList, "as", [elementDistributionSpec], ident ;

(* Variables *)
variableSection = "variable", {variableDecl}- ;
variableDecl =
    identifierList, "as", [elementDistributionSpec], ident ;

(* Streams *)
streamSection = "stream", {streamDecl}- ;
streamDecl = (ident, "is", pathName)
  | (ident, ':', pathNameList, "as", streamType) ;
streamType =
    [elementDistributionSpec], (ident | "connection") ;

(* Selectors *)
selectorSection = "selector", {selectorDecl}- ;
selectorDecl = identifierList, "as", selectorType ;
selectorType = [elementDistributionSpec], basicSelector ;
basicSelector =
    '(', identifierList, ')', ["default", ident] ;

(* Set *)
setSection = "set", {stmnt}- ;
domainSolutionMethods = '[', discretisationMethod, ','
  , expression, ',', expression, ']' ;
discretisationMethod =
    ("bfdm" | "cfdm" | "ffdm" | "ufdm" | "ocfem") ;

(* Boundary equations *)
boundarySection = "boundary", {stmnt}- ;

(* Equations *)
equationSection = "equation", {stmnt}- ;
```

```
(* - TASK entity ---------------------------------------- *)
taskEntity = "task", ident, {taskParameterSection}
  , {taskVariableSection}, {scheduleSection}, "end" ;

(* Parameters *)
taskParameterSection = "parameter", {taskParameterDecl}- ;
taskParameterDecl = identifierList, "as", taskParameterType ;
taskParameterType = basicTaskParameter | ("model", ident) ;
basicTaskParameter = basicType
  | "integer_expression"
  | "real_expression"
  | "logical_expression" ;

(* Variables *)
taskVariableSection = "variable", {taskVariableDecl}- ;
taskVariableDecl = identifierList, "as", basicType ;

(* Schedule *)
scheduleSection = "schedule", scheduledTask ;

(* - PROCESS entity ------------------------------------- *)
processEntity = "process", ident, {processElementDecl}
  , {monitorSection}, {setSection}, {equationSection}
  , {assignSection}, {presetSection}
  , {selectorAssignmentSection}, {initialConditionSection}
  , {processOptions}, {processSchedule}, "end" ;
processElementDecl = parameterSection
  | unitSection
  | randomSection ;

(* Random *)
randomSection = "random_stream", {randomStreamDecl} ;
randomStreamDecl = identifierList,(("as", "seed", expression)
  | ("as", "irreproducible")) ;

(* Monitor *)
monitorSection = "monitor", patternList ;
patternList = patternEntity, {',', patternEntity} ;
patternEntity = {patternBase}-, ';' ;
patternBase = ident | numLiteral | '(' | ')' | '*' | '%'
  | '.' | ',' ;

(* Assign *)
assignSection = ":=", {stmnt}- ;

(* Preset *)
presetSection =
    "preset", restorePreset, [{stmnt}-, restorePreset] ;
restorePreset = {simpRestore, [';']} ;
simpRestore =
    "restore", [storeTypeOption], strExpressionList ;
```

```
(* SelectorAssignment *)
selectorAssignmentSection = "selector", {stmnt}- ;

(* InitialCondition *)
initialConditionSection = ("initial", {stmnt}-)
  | ("initial", "steady_state") ;

(* ProcessOptions *)
processOptions = "solutionparameters", {execParaAssignment} ;
execParaAssignment =
    execParameter, ":=", simpleParaValue, [';'] ;
execParameter = ident | "monitor" ;
simpleParaValue =
    "initial" | expression | namedValAttributes ;
namedValAttributes =
    strLiteral, '[', [namedAttributeList], ']' ;
namedAttributeList = namedAttributeAssign
  , {(',' | ';'), namedAttributeAssign} ;
namedAttributeAssign = strLiteral, ":=", attributeValue ;
attributeValue = expression | namedValAttributes ;

(* Schedule *)
processSchedule = scheduleSection ;

(* - Statements ---------------------------------------- *)
stmnt = assignStmnt
  | equationStmnt
  | withinStmnt
  | forStmnt
  | ifStmnt
  | caseStmnt ;
assignStmnt = qualifiedName, ":=", setValues ';' ;
equationStmnt = [ident, ":"] additiveExpression
  , {("is" | '=' | "<>" | ">=" | '>' | "<=" | '<')
  , additiveExpression}- ';' ;
withinStmnt = "within", pathName, "do", {stmnt}-, "end" ;
forStmnt =
    "for", forInit, [forStep], "do", {stmnt}-, "end" ;
forInit = ident, ":="
  , additiveExpression, "to", additiveExpression ;
forStep = "step", additiveExpression, ;
ifStmnt = "if", expression, "then", {stmnt}-, "else"
  , {stmnt}-, "end" ;
caseStmnt = "case", pathName, "of", {caseClause}- "end" ;
caseClause =
    "when", pathName, ':', {stmnt}-, {caseSwitchTo} ;
caseSwitchTo =
    "switch", "to", qualifiedName, "if", expression, ';' ;

(* - Schedule operations ------------------------------- *)
scheduledTask = invocationTask
  | AssignmentTask
  | sequenceTask
  | parallelTask
```

```
  | whileTask
  | ifTask
  | continueTask
  | resetTask
  | replaceTask
  | reinitialTask
  | messageTask
  | switchTask
  | monitorTask
  | stopTask
  | saveTask
  | restoreTask
  | resetResultsTask
  | getTask
  | sendTask
  | sendMathInfoTask
  | lineariseTask
  | pauseTask ;
invocationTask =
    ident, ["(", taskParameterAssignList, ")"], [';'] ;
taskParameterAssignList =
    taskParameterAssign, {',', taskParameterAssign} ;
taskParameterAssign = ident, "is", expression ;
AssignmentTask = ident, ":=", setValues, ';' ;
sequenceTask = "sequence", {scheduledTask}-, "end" ;
parallelTask = "parallel", {scheduledTask}-, "end" ;
whileTask = "while", expression, "do", scheduledTask, "end" ;
ifTask = "if", expression, "then"
  , scheduledTask,["else", scheduledTask], "end" ;
continueTask = ("continue", "for", additiveExpression
  , [("and" | "or"), "until", expression], [';'])
  | ("continue", "until", expression, [';']) ;
resetTask = "reset", {stmnt}-, "end" ;
replaceTask =
    "replace", pathNameList, "with", {stmnt}-, "end" ;
reinitialTask =
    "reinitial", pathNameList, "with", {stmnt}-, "end" ;
messageTask = "message", strLiteral, [';'] ;
monitorTask = ("monitor", ident, [';'])
  | ("monitor", "frequency", expression, [';']) ;
stopTask = "stop", [';'] ;
switchTask = "switch", {stmnt}-, "end" ;
saveTask =
    "save", [storeTypeOption], strExpressionList, [';'] ;
restoreTask =
    "restore", [storeTypeOption], strExpressionList, [';'] ;
storeTypeOption = ('(', identifierList, ')')
  | identifierList ;
resetResultsTask = "resetresults", identifierList, [';'] ;
getTask = "get", [[foreignSignal], [foreignStatus]
  , {getEntity}-], "end" ;
getEntity = pathName, [":=", foreignPathName], ';' ;
sendTask = "send", [[foreignSignal], [foreignStatus]
  , {sendEntity}-], "end" ;
```

```
sendEntity = (foreignPathName, ":=", setValues, ';')
  | (pathName, ';') ;
lineariseTask = "linearise", [[foreignSignal]
  , [foreignStatus], ident, {linearEntity}-, ident
  , {linearEntity}-], "end" ;
linearEntity = [foreignPathName, ":="], pathName, ';' ;
pauseTask = "pause" [foreignSignal] [foreignStatus] ;
sendMathInfoTask =
    "sendmathinfo", [foreignSignal], [foreignStatus], [';'] ;
foreignSignal = "signalid", strLiteral ;
foreignStatus = "status", ident ;
foreignPathName = foreignPath, {DOT, foreignPath} ;
foreignPath = strLiteral, ['(', expressionList, ')'] ;

(* SetValues and expressionLists *)
setValues = domainSolutionMethods
  | ('[', expressionList, ']')
  | (expression, [':', [expression], ':', [expression]]) ;
expressionList = expressionInList, {',', expressionInList} ;
expressionInList = domainExpression | expression | ;
strExpressionList = strLiteral, {',', strLiteral} ;

(* - Expressions ---------------------------------------- *)
expression = logicalOrExpression ;
logicalOrExpression = logicalAndExpression
  , {"or", logicalAndExpression} ;
logicalAndExpression = relationalExpression
  , {"and", relationalExpression} ;
relationalExpression = additiveExpression
  , {('=' | "<>" | ">=" | '>' | "<=" | '<')
  , additiveExpression} ;
additiveExpression = multiplicativeExpression,
    {('+' | '-'), multiplicativeExpression} ;
multiplicativeExpression = unaryExpression,
    {('*' | '/' | "div" | "mod"), unaryExpression} ;
unaryExpression = ['+' | '-'] factorExpression ;
factorExpression = ("not", primaryExpression)
  | primaryExpression, [("|+" | "|-" | '|')
  | ('^', primaryExpression)] ;
primaryExpression = (constantExpression)
  | ('(', logicalOrExpression, ')')
  | ("time")
  | ("old", '(', logicalOrExpression, ')')
  | ("partial", '(', additiveExpression, ',',
    pathNameList, ')')
  | ("integral", '(', integralOperandList, ';',
    additiveExpression, ')')
  | ('$', '(', logicalOrExpression, ')')
  | (qualifiedName) ;
constantExpression = numLiteral | strLiteral | logicLiteral ;
integralOperandList =
    integralOperand, {',', integralOperand} ;
integralOperand = ident, ":=", domainExpression ;
```

```
domainExpression =
    additiveExpression, ':', additiveExpression ;

(* Qualified names *)
pathNameList = pathName, {',', pathName} ;
qualifiedName = (derivativeIdentifier)
  | (path, '.', derivativeIdentifier)
  | (pathName) ;
derivativeIdentifier = '$', pathEntity ;
pathName = path ;
path = pathEntity, {'.', pathEntity} ;
pathEntity = ident, ['(', expressionList, ')'] ;
identifierList = ident, {',', ident} ;

(* Identifier *)
ident = letter, {letter | digit} ;
letter =   'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'
  |'m'|'n'|'o'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|'A'
  |'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'
  |'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'X'|'Y'|'Z' ;
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;

(* Basic types *)
basicType = "integer" | "real" | "logical" ;
logicLiteral = "true" | "false" ;

(* Literals *)
numLiteral = {digit}, ['.', {digit}, [exponent]] ;
exponent = ('e'|'E'), ['+'|'-'], {digit}- ;
strLiteral = '"', {character - '"'}, '"' ;
```

Bibliography

[BN95] J. J. Barton and L. R. Nackman. Dimensional analysis. *C++ Report*, 7(1):39–40, 42–43, 1995.

[Buc14] E. Buckingham. On physically similar systems: illustrations of the use of dimensional equations. *Phys. Rev. Ser.*, 2(4):345–356, 1914.

[Car96] L. Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1):263–264, 1996.

[CG88] R. F. Cmelik and N. H. Gehani. Dimensional analysis with c++. *IEEE Software*, 5(3):21–27, 1988.

[DPL03] M. Sandberg D. Persson and B. Lisper. Automatic dimensional consistency checking for simulation specifications. In *44th Scandinavian Conference on Simulation and Modeling*, pages 13–18. SIMS, 2003.

[Fou22] J. B. Fourier. Theorie analytique de la chaleur. *Paris: Gauthier-Villars*, 1822.

[Hil88] P. N. Hilfinger. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, 1988.

[Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *The computer journal*, 26(4):366–374, 1983.

[JH99] S. P. Jones and J. Hughes. Report on the functional language haskell98, a non-strict, purely functional language. *http://haskell.org/definition/haskell98-report.pdf*, 1999.

[Jon99] M. P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, 1999.

[Ken94] A. J. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming*, volume 788, pages 348–362. Springer-Verlag, 1994.

[Ken96] A. J. Kennedy. Physical dimensions and programming languages. *Phd thesis, Univ. of Cambridge, UK*, 1996.

[Ken97] A. J. Kennedy. Relational parametricity and units of measure. pages 442–455. ACM, 1997.

[Kha01] R. Khanin. Dimensional analysis in computer algebra. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 201–208. ACM Press, 2001.

[KL78] M. Karr and D. B. Loveman. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, 1978.

[MTH89] R. Milner, M. Tofte, and R. Harper. The definition of standard ml. *MIT Press, Cambridge, Mass.*, 1989.

[New87] I. Newton. Philosophiae naturalis principia mathematica. *Streeter, London*, 1687.

[Nov95] G. S. Novak. Conversion of units of measurement. *Software Engineering, IEEE Transactions on*, 21(8):651–661, 1995.

[Ray78] Lord. Rayleigh. The theory of sound. *London*, 1878.

[Ray15] Lord. Rayleigh. The principle of similitude. *Nature*, 95(2368):66–68, 1915.

[Rit95] M. Rittri. Dimension inference under polymorphic recursion. In *7th Conference on Functional Programming Languages and Computer Architecture*, pages 147–159. ACM, 1995.

[WO91] M. Wand and P. M. O'Keefe. Automatic dimensional inference. *Computational Logic: in honor of J. Alan Robinson*, pages 479–486, 1991.